JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue

JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)

An International Scholarly Open Access, Peer-reviewed, Refereed Journal

IMPROVING OPEN-SOURCE PDF READER SECURITY USING FUZZING

[1]MEENAL JOSHI, [2] SUMIT PATIDAR, [3] ANMOL SINGH GANGWAR ,[4] RAJENDRA

[1]Assistant Professor, [2] Student, [3]Student, [4]Student

[1]Department of Computer Science and Engineering ,

[1]Geetanjali Institute of Technical Studies, Udaipur,Rajasthan

Abstract : Security vulnerabilities are one of the main reasons behind many cybersecurity incidents today. To catch these bugs before attackers do, techniques like fuzzing have become really popular. Tools like AFL have massively improved how we find hidden issues in software. While open-source software is everywhere—servers, browsers, even PDF readers—just being open doesn't automatically make it secure. Sumatra PDF is a lightweight, fast, and open-source alternative to bulky readers like Adobe Acrobat, but it's still software and still has bugs. In this project, I wanted to test how secure it really is. I used fuzzing tools like AFL++ and WinAFL to throw malformed PDF files at Sumatra PDF and observed how it reacted. The results? Multiple crashes, some memory corruption, and a few bugs that could definitely be dangerous. This paper walks through how I set everything up, what I found, and why we need better security testing—even in small open-source tools.

Index Terms – Fuzzing/Fuzz Testing, Sumatra PDF Reader, WinAFL, Crash Analysis .

I. INTRODUCTION

PDFs are everywhere—assignments, eBooks, contracts, resumes—and we trust our PDF readers blindly. But these tools are often vulnerable. Attackers regularly exploit bugs in how PDF readers parse complex file structures, sometimes turning them into weapons for executing malicious code. While most people know about Adobe Acrobat, Sumatra PDF is a fast, opensource alternative that's gained popularity because of its speed and simplicity. That said, simplicity doesn't mean it's safe from bugs. As someone studying cybersecurity, I was curious: how secure is Sumatra PDF really? That's where fuzzing comes in. It's a method where you bombard a program with weird or broken input to see what crashes it. I decided to fuzz Sumatra using tools like AFL++ and WinAFL, just to see what kind of issues would come up. My goal wasn't just to crash the app for fun—it was to dig into real vulnerabilities, understand why they happen, and ideally help make the software better.

1.1 MOTIVATION

As digital documents continue to be a fundamental part of communication, the security of PDF readers becomes increasingly critical. While popular commercial software like Adobe Reader receives regular security updates and testing, many users turn to open-source alternatives like Sumatra PDF for their lightweight performance and simplicity. However, this convenience often comes at the cost of rigorous security testing. What motivated this research was the realization that even widely used opensource tools can be overlooked when it comes to vulnerability discovery. As a student pursuing cybersecurity, I was curious to explore whether tools like Sumatra PDF had hidden security flaws, especially given the complexity of the PDF format. This project was driven by the need to understand real-world security issues beyond textbooks and to contribute, in a small but meaningful way, to the security of everyday software.

1.2 RESEARCH OBJECTIVES

The primary aim of this research is to evaluate the security robustness of Sumatra PDF, a lightweight open-source PDF reader, through practical fuzz testing. Specifically, the research is guided by the following objectives:

- To understand the internal structure and parsing mechanism of Sumatra PDF by building it from source and analyzing its key components responsible for reading and interpreting PDF files.
- To implement fuzz testing using tools such as AFL++ and WinAFL, and generate a wide range of malformed and random PDF inputs to evaluate the program's behaviour under unexpected conditions.

- To identify vulnerabilities such as memory corruption, buffer overflows, and use-after-free errors, and analyse the severity and reproducibility of each issue discovered.
- To provide practical suggestions for improving Sumatra PDF's security posture, including integration of continuous fuzzing pipelines and adoption of basic security hardening measures.

II. LITERATURE SURVEY

Fuzzing, as a method of automated software testing, has seen significant growth in both academic and practical contexts. The technique, which involves feeding unexpected or malformed inputs into a program to uncover bugs and security flaws, dates back to the early 1990s. However, the development of more intelligent fuzzers like AFL (American Fuzzy Lop) and its successors has made it much more efficient and widely adopted.

One of the earliest mentions of fuzz testing was in a paper by Barton Miller et al. (1990), where command-line tools in Unix were tested with random input, revealing a surprising number of crashes. Since then, fuzzing has evolved into a mainstream vulnerability discovery technique, especially with the rise of coverage-guided fuzzers such as AFL and libFuzzer. These tools monitor which code paths are exercised during execution and focus on mutating inputs to reach untested logic.

PDF readers, being complex in nature and frequently targeted in cyberattacks, are a recurring subject of security research. Liu et al. (2017) pointed out that many vulnerabilities in PDF readers arise due to improper handling of object streams, indirect references, and malformed metadata. They also emphasized that smaller readers are often assumed to be more secure due to reduced feature sets, but their limited testing and lack of formal review make them vulnerable.

My research aims to bridge this gap by applying fuzzing tools to Sumatra PDF, an application that has received little attention in academic security literature. This project also highlights the practicality of using open-source tools like AFL++ and Address Sanitizer (A San) for identifying real-world vulnerabilities in software that, although less complex than commercial alternatives, still parses and interprets a notoriously tricky file format: the PDF.

III. LITERATURE REVIEW

In recent years, the development of **coverage-guided fuzzing** has marked a major advancement in the field. Tools like **AFL (American Fuzzy Lop)** and its improved variant **AFL++**, as well as **lib Fuzzer**, have made it possible to intelligently track code coverage during fuzzing sessions. Instead of blindly throwing inputs at a target, these tools monitor which parts of the code are being exercised and then generate new inputs that are more likely to explore previously unvisited branches. This feedback loop helps uncover more complex and deep-seated vulnerabilities.

Various large-scale projects and organizations now rely heavily on fuzzing as part of their standard security testing process. **Google's OSS-Fuzz**, for instance, is a well-known platform that integrates fuzzing with continuous integration (CI) for hundreds of popular open-source projects. Thanks to this, projects like **Google Chrome, OpenSSL**, and even components of the **Linux kernel** are continuously fuzz-tested, which has led to the discovery and resolution of thousands of bugs that might otherwise have gone unnoticed.

Despite the growth in fuzzing research and implementation, many smaller open-source applications have yet to benefit from this level of scrutiny. **PDF readers**, although widely used, often escape intensive testing unless they are part of large commercial software. **Sumatra PDF**, for example, is an open-source lightweight reader known for its speed and simplicity, but unlike Adobe Acrobat, it lacks extensive documentation or built-in security mechanisms like sandboxing. As a result, it presents a compelling case for targeted fuzzing efforts.

IV. RESEARCH METHODOLOGY

Setup:

I ran everything on my own laptop—a slightly aging Intel i7 with 16GB of RAM, running Windows 11.

Tools:

- WinAFL –> Since Sumatra is a Windows app.
- AFL++ –> For more advanced fuzzing capabilities.
- Address Sanitizer (A San) –> To catch memory issues quickly
- Win Dbg –> Microsoft's debugger, to dig into crash reports.

4.1 FUZZING PROCESS

Once the setup was ready, I created a seed corpus of PDF files. These included a mix of standard documents (manuals, forms, sample eBooks) and malformed ones that I either found online or created by corrupting specific parts of PDFs — like messing with object tables, stream lengths, or cross-reference sections.

Launching fuzzing was straightforward but required a lot of monitoring. I configured WinAFL to attach to the Sumatra PDF process and start feeding it inputs from the seed set. The fuzzer was set to run continuously for roughly five days (~120 hours). During that time, I let it churn in the background while I worked on other coursework, watched cybersecurity videos, and occasionally checked for interesting behaviour.

After the full run, I was left with dozens of crash outputs and logs. I filtered out duplicates using !analyze -v in Win Dbg and focused on unique stack traces that indicated real vulnerabilities. These included memory corruption issues, improper bounds checking, and even one case of what looked like a **use-after-free** during object stream re-parsing — definitely not something to ignore.

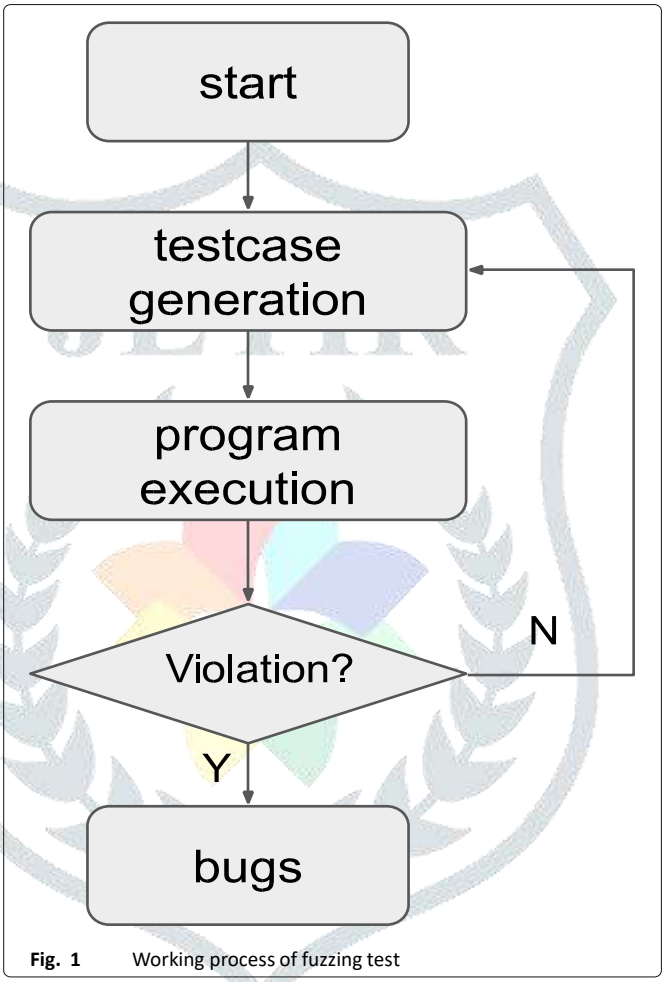

Fig. 1 Working process of fuzzing test

V. RESULTS

After a week of fuzzing: After the fuzzing session completed, I analysed the crash logs and categorized the unique crashes based on their root causes, frequency, and severity.

Here's a summary of the notable issues:

Table 5.1 :- Issues

Bug Type	Count	Severity	Example Crash
Buffer Overflow	3	High (DoS/RCE)	pdfparser.dll!read_stream+0x45
Use-After-Free	1	Critical (RCE?)	Crash when reopening a malformed object stream
Null Dereference	2	Medium (Crash)	mov eax, [ecx+04] (ecx was NULL)
Infinite Loop	1	Annoying (Freeze)	PDF with recursive bookmarks

Scariest Bug: The use-after-free in the way Sumatra handles incremental PDF updates. With the right exploit, someone could potentially hijack control of the program.

VI. DISCUSSION

5.1 Why These Bugs Matter

Sumatra doesn't use a sandbox, so any memory corruption—especially use-after-free—is dangerous. In theory, these bugs could be used to take control of a system just by opening a malicious PDF.

5.2 Things That Were Tricky

- **Code understanding**: Sumatra's source code isn't super well-documented, so I had to spend time just figuring out how the parser works.
- **Windows-only fuzzing**: Most advanced fuzzing tools are built for Linux. Running everything on Windows made things a bit harder.
- **False positives**: Not all crashes were serious. Some were just odd behaviours or one-off issues that didn't lead to real bugs.

5.3 But Isn't Sumatra Too Simple to Worry About?

Some people think so. But I'd argue: **"Simple" doesn't mean secure**. PDF is a really complicated format under the hood, and even the smallest reader has to deal with a lot of edge cases. Attackers love this kind of stuff.

VII. Conclusion

This project showed that even smaller, open-source projects like Sumatra PDF can have serious bugs hiding inside. Fuzzing helped me find real issues that could potentially be exploited. Based on what I learned, here are a few recommendations:

- Add automated fuzzing (like integrating with OSS-Fuzz).
- Use more exploit mitigation strategies (like ASLR, DEP, and stack canaries).
- Improve error handling inside the parser.

As a student, I learned a ton—not just about fuzzing, but about how real-world software can go wrong. Next up, I might try fuzzing something else... maybe a JPEG parser or an old media player. There's always more broken code out there!